Tutorial 9

- Introduction – Image processing overview
- 2d convolution – example on board
- Convolutional Neural Networks

Tomer Golany – tomer.golany@gmail.com
Introduction
Image classification is the task of taking an input image and outputting a class (a cat, dog, etc) or a probability of classes that best describes the image.

For humans, this task of recognition is one of the first skills we learn from the moment we are born and is one that comes naturally and effortlessly as adults.
Image Representation

- When a computer sees an image (takes an image as input), it will see an array of pixel values.

- An image is a **continues 2d-signal**. When we represent an image in a computer, we sample the 2d signal at some frequency and get a **discrete** 2d-signal.

- The resolution and the size of the image determines the size of the array of pixels.

- For a black & white (gray scale) image the representation is with one 2d-array of size height X width.

- For a color image (RGB) the representation is with 3 2d-arrays, each one corresponds to different color (Red, Green, Blue, each one is called a channel).
Each element $a(i)(j)$ in the 2d-array is a **pixel**, and it represents the **intensity** of the image at the location $i,j$ of the image.

A value of a pixel can be between 0 (=Black) – 255 (=White). (we can also normalize to be between 0 - 1)

If the image is black & white, then each value between 0-255 is a gray level intensity.

If the image is RGB then for each channel, the values are the Blue/Green/Red intensity.
Images as functions:

- We can think of an image as a function, \( f \), from \( \mathbb{R}^2 \rightarrow \mathbb{R} \):

- \( f(x, y) \) gives the intensity at position \((x, y)\)

- A color image is just three functions pasted together. We can write this as:

\[
 f(x, y) = \begin{bmatrix} r(x, y) \\ g(x, y) \\ b(x, y) \end{bmatrix}
\]
Image Convolution
Image convolution

- In image processing, a **kernel, convolution matrix, filter** or **mask** is a small Matrix.

- It is used for blurring, sharpening, edge detection, and more.

- This is accomplished by doing a **convolution** between a **kernel** and an **image**.

- Depending on the element values, a kernel can cause a wide range of effects.
Image convolution

- Mathematical definition of 2d-convolution:

\[
f[x,y] * g[x,y] = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} f[n_1,n_2] \cdot g[x-n_1,y-n_2]
\]

- Convolution is an operation on two functions \(f\) and \(g\), which produces a third function that can be interpreted as a "filtered" version of \(f\).

- If \(f\) is defined on a spatial variable like \(x\) rather than a time variable like \(t\), we call the operation *spatial convolution*. 
Calculating 2d convolution - Example

- Example of convolution of 3x3 input image and kernel with the same size:

- To make the result the dimension as the input image, we will pad with zeros the input image.
Calculating 2d convolution - Example

- First, flip the kernel, which is the shaded box, in both horizontal and vertical direction.

- Move the filter over the input array. If the kernel is centered (aligned) exactly at the sample that we are interested in, multiply the kernel data by the overlapped input data.

- The accumulation (adding these 9 multiplications) is the last thing to do to find out the output value.
## Kernel Examples

<table>
<thead>
<tr>
<th>Operation</th>
<th>Kernel</th>
<th>Image result</th>
</tr>
</thead>
</table>
| **Identity**   | \[
|                | \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} | |
| **Edge detection** | \[
|                | \begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix} | |
|                | \[
|                | \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} | |
|                | \[
|                | \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} | |
## Kernel Examples

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Matrix</th>
<th>Example Image</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sharpen</strong></td>
<td>$\begin{bmatrix} 0 &amp; -1 &amp; 0 \ -1 &amp; 5 &amp; -1 \ 0 &amp; -1 &amp; 0 \end{bmatrix}$</td>
<td><img src="image1.jpg" alt="Image" /></td>
</tr>
<tr>
<td><strong>Box blur</strong></td>
<td>$\frac{1}{9} \begin{bmatrix} 1 &amp; 1 &amp; 1 \ 1 &amp; 1 &amp; 1 \ 1 &amp; 1 &amp; 1 \end{bmatrix}$</td>
<td><img src="image2.jpg" alt="Image" /></td>
</tr>
<tr>
<td>(normalized)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Gaussian blur 3 \times 3</strong></td>
<td>$\frac{1}{16} \begin{bmatrix} 1 &amp; 2 &amp; 1 \ 2 &amp; 4 &amp; 2 \ 1 &amp; 2 &amp; 1 \end{bmatrix}$</td>
<td><img src="image3.jpg" alt="Image" /></td>
</tr>
<tr>
<td>(approximation)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Filters as feature detection:

- Filters can also be used as **feature identifiers**
- When I say features, I’m talking about things like straight edges, simple colors, and curves.
- For example lets say that all images we want to classify have in common with each other a certain curve.
- We can define the following filter as the **curve detector**:
- As a curve detector, the filter will have a pixel structure in which there will be **higher numerical values along the area that is a shape of a curve.**
Now let’s take an example of an image that we want to classify, and let’s put our filter at the top left corner. Basically, in the input image, if there is a shape that generally resembles the curve that this filter is representing, then all of the multiplications summed together will result in a large value!
Now let’s see what happens when we move our filter:

- The value is much lower!
- This is because there wasn’t anything in the image section that responded to the curve detector filter.

The output of the convolution is an **activation map**. The activation map will show the areas in which there are mostly likely to be curves in the image.
Convolutional Neural Networks
The Task

- Back to Deep Learning.
- The Task: to make the job really simple, we’ll only try to recognize the number “8” in a data set of images.
- We will use the MNIST data set - MNIST provides 60,000 images of handwritten digits, each as an 18x18 image:
First try: Fully connected

- Why not using feed-forward network?

- We can flatten the input 2d array (which represents the image) to 1d input vector.

- To feed an image into fully connected neural network, we simply treat the 18x18 pixel image as an array of **324 numbers**

- So the input vector will have 324 features, that means that each neuron at the first layer will need to have **324 weights**.
Problems with fully connected on images

- We might need large number of weights which will cause the training process to be slow.

- The NN in the example on the MNSIT images will learn good to recognize “8” on simple images where the letter is right in the middle of the image.

- Our “8” recognizer totally fails to work when the letter isn’t perfectly centered in the image. Just the slightest position change ruins everything:

  This is because our network only learned the pattern of a perfectly-centered “8”. It has absolutely no idea what an off-center “8” is. It knows exactly one pattern and one pattern only.
Second Try: Searching with a Sliding Window

- What if we just scan all around the image for possible “8”s in smaller sections, one section at a time, until we find one?

- It’s the brute force solution. It works well in some limited cases, but it’s really inefficient.

- We have to check the same image over and over looking for objects of different sizes. We can do better than this!
Third Try: More Data

- When we trained our network, we only showed it “8”s that were **perfectly centered**.

- What if we train it with more data, including “8”s in all different positions and sizes all around the image?

- We don’t even need to collect new training data. We can just write a script to generate new images with the “8”s in all kinds of different positions in the image:
Using this technique, we can easily create an endless supply of training data.

More data makes the problem harder for our neural network to solve, but we can compensate for that by making our network bigger and thus able to learn more complicated patterns.

That still isn’t going to get us all the way to a solution. We need to be smarter about how we process images into our neural network.

There should be some way to make the neural network smart enough to know that an “8” anywhere in the picture is the same thing without all that extra training. Luckily... there is!
Intro to NLP and Deep Learning

Convolutional neural networks

The solution is convolution:
Step 1: Break the image into overlapping image tiles

Similar to our sliding window search above, let’s pass a sliding window over the entire original image and save each result as a separate, tiny picture tile:

By doing this, we turned our original image into 77 equally-sized tiny image tiles.
Step 2: Feed each image tile into a small neural network

Earlier, we fed a single image into a neural network to see if it was an “8”. We’ll do the exact same thing here, but we’ll do it for each individual image tile:

However, there’s one big twist:

We’ll keep the same neural network weights for every single tile in the same original image.

In other words, we are treating every image tile equally.

If something interesting appears in any given tile, we’ll mark that tile as interesting.
Intuitive Explanation

- **Step 3: Save the results from each tile into a new array:**
- We don’t want to lose track of the arrangement of the original tiles.
- So we save the result from processing each tile into a grid in the same arrangement as the original image. It looks like this:
### Step 4: Downsampling

The result of Step 3 was an array that maps out which parts of the original image are the most interesting. But that array is still pretty big, (or even same size, if we padded the image).

To reduce the size of the array, we **downsample** it using an algorithm called **max pooling**.

We’ll just look at each 2x2 square of the array and keep the biggest number: The idea here is that if we found something interesting in any of the four input tiles that makes up each 2x2 grid square, we’ll just keep the most interesting bit. This reduces the size of our array while keeping the most important bits.
Final Step: Make a prediction:

- we’ve reduced the image down into a fairly small array.
- we can use that small array as input into another neural network. This final neural network will decide if the image is or isn’t a match.
From the lectures:
Convolution Layer

32x32x3 image

5x5x3 filter

Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”
**Convolution Layer**

- Filters always extend the full depth of the input volume.
- A 32x32x3 image is convolved with a 5x5x3 filter.
- Convolve the filter with the image i.e. “slide over the image spatially, computing dot products.”
Convolution Layer

32x32x3 image

5x5x3 filter $\mathcal{W}$

1 number: the result of taking a dot product between the filter and a small 5x5x3 chunk of the image (i.e. $5*5*3 = 75$-dimensional dot product + bias)

$$w^T x + b$$
Convolution Layer

32x32x3 image
5x5x3 filter

convolve (slide) over all spatial locations

activation map
Consider a second, green filter.

Convolution Layer

32x32x3 image
5x5x3 filter

Convolve (slide) over all spatial locations

Activation maps
For example, if we had 6 5x5 filters, we’ll get 6 separate activation maps:

We stack these up to get a “new image” of size 28x28x6!
A closer look at spatial dimensions:

7x7 input (spatially) assume 3x3 filter
A closer look at spatial dimensions:

7x7 input (spatially) assume 3x3 filter
A closer look at spatial dimensions:

7

7x7 input (spatially) assume 3x3 filter
A closer look at spatial dimensions:

7x7 input (spatially) assume 3x3 filter
A closer look at spatial dimensions:

7x7 input (spatially)  
assume 3x3 filter

=> 5x5 output
A closer look at spatial dimensions:

7x7 input (spatially) assume 3x3 filter applied with stride 2
A closer look at spatial dimensions:

7x7 input (spatially) assume 3x3 filter applied with stride 2
A closer look at spatial dimensions:

7x7 input (spatially) assume 3x3 filter applied with stride 2 => 3x3 output!
A closer look at spatial dimensions:

7x7 input (spatially) assume 3x3 filter applied with stride 3?
A closer look at spatial dimensions:

7x7 input (spatially) assume 3x3 filter applied with stride 3?

doesn’t fit! cannot apply 3x3 filter on 7x7 input with stride 3.
Output size: 
\[(N - F) / \text{stride} + 1\]

e.g. \(N = 7, F = 3\):
- \(\text{stride } 1 \Rightarrow (7 - 3)/1 + 1 = 5\)
- \(\text{stride } 2 \Rightarrow (7 - 3)/2 + 1 = 3\)
- \(\text{stride } 3 \Rightarrow (7 - 3)/3 + 1 = 2.33\)
In practice: Common to zero pad the border

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

e.g. input 7x7

3x3 filter, applied with **stride 1**

pad with **1 pixel** border => what is the output?

7x7 output!
In practice: Common to zero pad the border

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- e.g. input $7 \times 7$
- $3 \times 3$ filter, applied with **stride 1**
- **pad with 1 pixel** border => what is the output?

**7x7 output!**

In general, common to see CONV layers with stride 1, filters of size $FxF$, and zero-padding with $(F-1)/2$. (will preserve size spatially)

- e.g. $F = 3$ => zero pad with 1
- $F = 5$ => zero pad with 2
- $F = 7$ => zero pad with 3
### MAX POOLING

A single depth slice of a convolutional layer is shown. The slice contains values:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

The max pooling operation is applied with 2x2 filters and a stride of 2. The resulting pool is:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

The max is taken within each 2x2 block of the input slice.
Pooling Layer Summary

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires three hyperparameters:
  - their spatial extent $F$,
  - the stride $S$,
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F) / S + 1$
  - $H_2 = (H_1 - F) / S + 1$
  - $D_2 = D_1$

Common image processing settings:

$F = 2, S = 2$
$F = 3, S = 2$